Stable and Accurate Network Coordinates

Stable and Accurate Network Coordinates

Jonathan Ledlie
and
Margo Seltzer

TR-17-05

Computer Science Group
Harvard University
Cambridge, Massachusetts

Stable and Accurate Network Coordinates

Jonathan Ledlie and Margo Seltzer

***Abstract*— Synthetic coordinate systems that mirror latencies between physical hosts have become a part of the toolbox networking researchers would like to use in real deployments. However, the most promising algorithm for building these coordinate systems, Vivaldi, breaks down when run under real world conditions. Previous work on network coordinates has examined their performance in simulation through the use of a latency matrix, which summarizes each link with a single latency. In a deployment, instead of perceiving a single latency for each link, nodes see a stream of distinct observations that may vary by as much as three orders-of-magnitude. With no means to discern an appropriate latency for each link, coordinate systems are prone to high error and instability in live deployments.**

Two simple enhancements improved Vivaldi's accuracy by 54% and coordinate stability by 96% when run on a real large-scale network. First, we use a non-linear low pass filter to ascertain a clear underlying signal from each link. These filters primarily improve accuracy. Second, we introduce a distinction between system- and application-level coordinates. We evaluate a set of change-detection heuristics that allow coordinates to evolves at the system-level and only initiate an application-level update after a coordinate has undergone a significant change. These application-level coordinates retain the filter's high accuracy and dramatically increase coordinate stability.

***Index Terms*— Simulations, Stochastic processes/Queueing theory, Experimentation with real networks/Testbeds.**

I. Introduction

VIVALDI, a simple decentralized algorithm, embeds nodes in a network into a relative coordinate system [4], [5]. Coordinate systems are useful in a wide range of contexts, including large scale content distribution, routing, and stream-based overlay networks [10], [2], [20]. We used the Vivaldi algorithm to create a coordinate system with hundreds of nodes on the Internet as part of our work on stream-based overlay networks [19].

Yet, when run on a live system, the original algorithm does not produce stable, accurate coordinates. The discrepancy between what we found and the results from the original paper is primarily a result of the orders-of-magnitude variation in latency measurements between the same pairs of nodes that actually occur when running a coordinate system on a real network: inter-node latencies were fixed using a derived latency matrix in the original set of experiments. A few simple changes to the algorithm produced coordinates that are stable, accurate, and adapt to changing network conditions. This paper describes these modifications and how to create a relative coordinate system under "real world" conditions.

In Section II, we explain what the Vivaldi algorithm is and how to measure it, emphasizing the importance of coordinate stability for applications. After this section, the paper makes the following contributions:

- In Section III, we examine a latency distribution that exemplifies a typical input and discuss why the original algorithm experiences difficulty when used without a static latency matrix.
- In Section IV, we present a simple method for stabilizing coordinates by keeping a small history of samples with each node. This method improves both coordinate stability and accuracy; however, coordinate stability remains at a level unacceptable to most applications.
- In Section V, we differentiate between application- and system-level coordinates and compare four heuristics for improving application-level stability while maintaining accuracy. We find that when we insert a sliding window-based mechanism for change-detection borrowed from the database literature, an application's view of its network coordinates becomes significantly more stable.
- In Section VI, we build histories and application-level coordinates into an implementation that we run on a large network, resulting in a 54% improvement in accuracy and a 96% improvement coordinate stability.

In Section VII we discuss related work and in Section VIII we conclude.

II. Vivaldi Algorithm

The Vivaldi algorithm provides a simple, lightweight method for participants in a distributed system to form a Euclidean metric space, where the distance between any two nodes is an estimate of their true latencies. To the best of our knowledge, Vivaldi is the only completely distributed coordinate formation algorithm that requires neither well-known landmarks nor significant computation. The algorithm exhibits two useful properties for distributed systems:

- Two nodes do not need to have communicated previously for the latency between them to be estimated. Therefore, the algorithm scales to thousands or millions of nodes.
- The algorithm continues to refine coordinates as the true network conditions change over time. For example, if the latency of a link changes due to a BGP route change, coordinates adjust and restabilize quickly.

While these two properties are exhibited by the Vivaldi algorithm only in theory, it is nonetheless important that our methods for increasing its stability and accuracy do not fundamentally alter these properties in practice.

Vivaldi models the network as a collection of springs that pull on each node's coordinate. The original algorithm works as follows. Each node retains its coordinate $\overrightarrow{x_i}$ and its confidence in this coordinate $w_i \in (0, 1)$. All coordinates are the same low dimension, which is fixed *a priori*. Nodes adjust

J. Ledlie and M. Seltzer are with the Division of Engineering and Applied Science, Harvard University, Cambridge, MA. E-mail: {jonathan,margo}@eecs.harvard.edu .

VIVALDI$(l_{ij}, \overrightarrow{x_j}, w_j)$
1 $w_s = \frac{w_i}{w_i + w_j}$
2 $\epsilon = \frac{|\|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij}|}{l_{ij}}$
3 $\alpha = c_e \times w_s$
4 $w_i = (\alpha \times \epsilon) + ((1 - \alpha) \times w_i)$
5 $\delta = c_c \times w_s$
6 $\overrightarrow{x_i} = \overrightarrow{x_i} + \delta \times (\|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij}) \times u(\overrightarrow{x_i} - \overrightarrow{x_j})$

Fig. 1. *Vivaldi* update algorithm. u is the unit vector function.

their coordinates and confidences through observations of their latencies to other nodes in the system. These observations can be explicit pings or may be gleaned from existing traffic. Through successive samples, each node refines its coordinates and increases its confidence. Like a network of springs, coordinates become more accurate and stable with each successive adjustment.

Each node updates its coordinate and confidence with each new latency observation based on the pseudocode shown in Figure 1. An observation consists of the remote node's coordinate $\overrightarrow{x_j}$, its confidence w_j, and a new measurement of the latency between the two nodes, i and j. First, a weight w_s is assigned to this observation based on how confident nodes i and j are relative to one another (Line 1). In essence, this allows more confident nodes to tug harder than less confident ones. Second, they find how far off the observation was from what was expected based on the coordinates; this is the relative error ϵ of this measurement (Line 2). Third, node i updates its confidence w_i with an exponentially-weighted moving average. Unlike most EWMAs, however, the α, or weight given to the current observation, is not fixed. Instead it is weighted according to how much trust is given to the current observation (Lines 3-4). If this causes node i's confidence to go above one or below zero, it is forced to remain in bounds (not shown). Lines 5-6 update the coordinate. Also based on the confidence of nodes i and j, δ is the pull of this observation on the coordinate. In line 6, δ dampens the magnitude and direction of the change applied to the coordinate. Constants c_e and c_c affect the maximum change an observation can have on confidence and coordinates, respectively. They have the same effect as the tuning parameter in a standard exponentially-weighted moving average (EWMA): a low value of 0.05, for example, limits the weight given to any new observation and a high value of 0.25, for example, causes faster adjustments to new observations. Larger values for α may weigh outliers too heavily. We found any setting of c_c and c_e in this range to have minimal impact on large scale behavior. We used $c_c, c_e = 0.25$, which are the same values used in the original authors' Vivaldi simulator [7].

Bootstrapping the algorithm is simple. Coordinates are initially set to the origin. Each node stores a list of *neighbors*, *i.e.*, nodes that it samples. It is assumed that a node knows at least one other node when it enters the system. In our implementation, nodes learn new neighbors by attaching the address of one other node to each sampling message, *i.e.*, through gossip, and sample their neighbors in round-robin order.

Instead of using a pure metric space, Vivaldi can be modified to include a *height* h, which changes the distance between nodes i, j to $\|\overrightarrow{x_i} - \overrightarrow{x_j}\| + h_i + h_j$. The purpose of *height* is to capture the latency of the access link, while the coordinates themselves capture the long-haul links. Because our larger project [19] and the growing body of work using network coordinates use pure metric spaces (*e.g.* [13], [1], [9]), we did not include a *height*, although the techniques we present would allow for their use. We present results using three dimensions.

A. Measuring Coordinate Systems

In this context, accuracy is measured by comparing the difference between the expected and actual latencies for an observation. The error of a link for a particular observation l_{ij} is:

$$e = | \ \|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij} \ |$$

Depending on context, the accuracy for the system is the sum of these quantities for all nodes, the sum of the error squared (the mean squared error), or the median for each node. Accuracy can also be normalized by dividing by l_{ij}; this *relative error* is the same quantity as ϵ in Figure 1. We use relative error as the metric of accuracy because it facilitates comparison of a wide range of latencies.

Note that l_{ij} is a time dependent quantity because inter-node latencies are not fixed nor does the same link provide the same result with each observation. Instead of being a single quantity, L_{ij} is actually a distribution that depends on the characteristics of the link. One can consider the distribution L_{ij} the *true latency*. The original evaluation assumed that links returned the same measurement each time; in other words, that all l_{ij}'s were equal for a given link.

We measure *per-node* relative error instead of *per-link* relative error. The distribution of per-node relative error is the collection of errors for each node for all of its observations. Measuring per-link error would assume that a static, scalar latency matrix exists against which we could compare coordinates after a number of iterations. Because our underlying network is changing, this matrix, and hence this metric, cannot be computed.

Stable coordinates are particularly important when an application is using Vivaldi and a coordinate change triggers application activity. A stable coordinate system is one in which coordinates are not changing over time, assuming that the network itself is unchanging. Thus, links may produce some distribution of observations, but as long as this distribution does not change, neither should stabilized coordinates. We use the rate of coordinate change

$$s = \frac{\sum \Delta \overrightarrow{x_i}}{t}$$

to quantify stability. Our metric space, the numerator, is in milliseconds and we measure change in this space in seconds; thus, stability is in ms/sec unless otherwise noted.

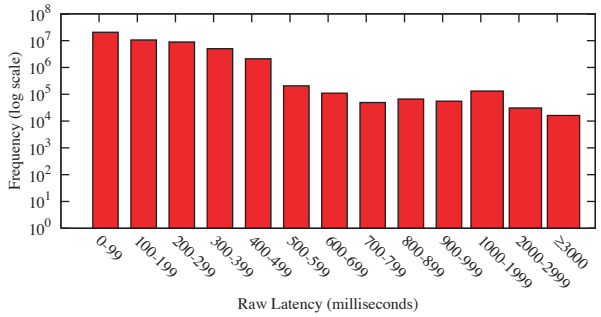

Fig. 2. Frequency histogram of raw latency measurements between 269 PlanetLab nodes.

III. Latency Measurements

When we first implemented Vivaldi, we found that lone samples, often orders-of-magnitude greater than expected, would periodically distort the entire coordinate system. These instabilities resulted when raw latency data was fed into the algorithm.

An examination of a set of raw latency data shows rare but persistent samples orders-of-magnitude larger than the common case. We collected a set of latency data from 269 PlanetLab [17] nodes over three days starting May 2, 2005, totaling 43 million samples. PlanetLab is a collection of approximately 500 machines spread around the world, located primarily at universities and research labs. To gather the trace, each node measured the latency to another node with an application-level UDP ping once per second. We used application-level pings because we intend to eventually use measurements of existing traffic as input, rather than extra explicit pings.

We summarize the total distribution of measurements in Figure 2. The data show that 0.4% of the measurements are greater than one second, which is longer than the common case even for inter-continental links. Instead of a steady stream of measurements, the fact that many measurements are above the largest expected latency suggests that many links may be experiencing serious delays that Vivaldi must automatically incorporate. The broad range of measurements severely curtails accuracy and stability.

We examined individual links to confirm that they too exhibited similar behavior. Not only did the entire distribution have a long tail, with most links below several hundred milliseconds, but individual links had as well. Figure 3 illustrates one representative link. It shows that a significant number of observations extend beyond the median (Figure 3, top) and that these infrequent order-of-magnitude delays are spread over time (Figure 3, bottom).

Because of the long tail, the mean of the raw values would not be a good predictor for future observations. Instead, the expected latency appeared to be predictable by taking a low percentile of some portion of the previous observations. This expected latency is a better measure of what Vivaldi should use as its approximation of the link latency, not the raw values. Instead of feeding raw observations into Vivaldi, we would filter the input data to remove the heavy tail. By giving Vivaldi a steadier input, our goal is for each link to experience lower relative error and greater stability by exhibiting less coordinate change over time.

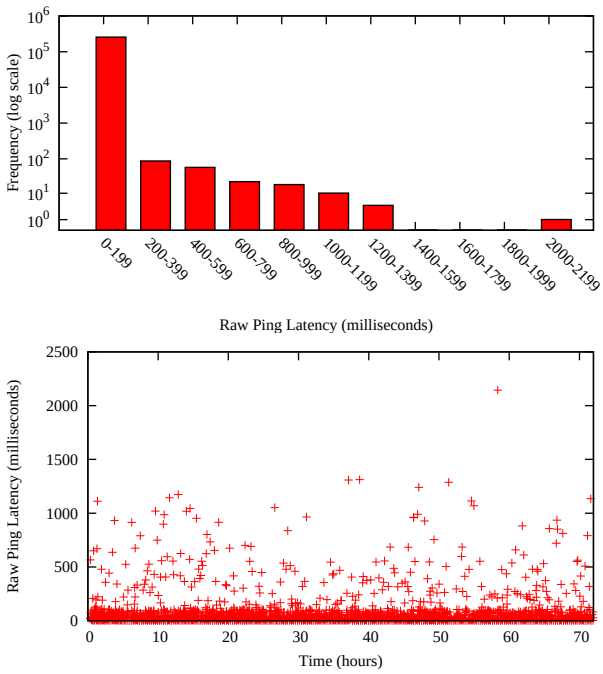

Fig. 3. Histogram and scatterplot of raw latency measurements from one PlanetLab node to another during a three day trace. Measurements vary by two orders-of-magnitude. Long latency pings continue to occur throughout the trace.

IV. Filtering with Histories

Based on our analysis of link latencies, a percentile of some window of previous observations appeared to be a good predictor of future values. Statistically, this is known as a Moving Percentile (MP) filter, a variant on the Moving Median filter, and has been used to filter out heavy-tailed error in other disciplines (*e.g.* [8], [14]). It is a non-linear filter, which removes non-Gaussian noise and lets through low frequencies. MP filters exhibit edge preservation and are robust against noise impulses. A MP filter has two parameters: (1) the size h of the history window and (2) the percentile p returned as the prediction for the next observation.

To examine the predictive effectiveness of the MP filter with different parameters, we examined how the filter performed on each link from the PlanetLab trace. Each link consists of a series of observations; the relative error is the difference between the filter's prediction and the next observation, divided by the next observation.

We ran an experiment in which we varied the size of the window and the percentile used to surmise the next value. Using the three day trace, we applied different filters to predict what the next observation would be and calculated the relative error between each prediction and the true observation. We plot the relative error for all of the links in the system as we vary the history size h and keep $p = 25$ in Figure 4. The results show that a history of only four observations achieves

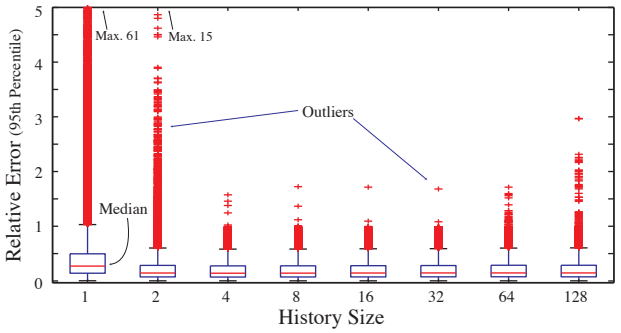

Fig. 4. Short histories of previous observations are sufficient to reduce the error in predicting the next latency observation. The boxplots show the relative error of all of the links in the system. They show that filters based on the most recent four observations predict with the least error.

the best performance (lowest error) with the fewest outliers. Using $p = 25$, the minimum with a history of four, resulted in slightly lower error than $p = 50$ for the MP filter.

Although long histories do not perform substantially worse, intuitively it makes sense that longer histories do not perform better: they are slow to adjust to any changes in network conditions. That short histories perform well is good for three reasons: (1) they can be acquired through fewer rounds of observations, (2) they require less state, and (3) they will be quickest to adjust to any latency shifts.

A. Vivaldi with the MP Filter

In order to compare Vivaldi with and without the MP filter, we built a simulator that accepted our raw ping trace as input and mimicked the distributed behavior of Vivaldi. Through a comparison of running Vivaldi on a real network and in our simulator, we found the simulator provided a high degree of verisimilitude.

Using the simple MP filter substantially improves both the accuracy and stability metrics. With the parameters that showed the best ability to predict subsequent samples — taking the 25^{th} percentile (minimum) of the previous four observations — we compared Vivaldi with and without MP filtering. We ran Vivaldi on a four hour section of the trace and show cumulative distributions for the second half of the run, eliminating start-up effects (we examine the rate of start-up in Section V). We measured per-node accuracy and system-wide stability and summarize the results in Figure 5. The data show that the MP filter at least doubles accuracy and stability for most nodes. Its primary benefit, however, is that it eliminates the periodic distortion of the entire coordinate space that occurs with no filtering. This is shown through the reduction of the long tail of instability by three orders-of-magnitude. In the application we developed, these distortions would cause a cascade of other updates to occur and using the MP filter ameliorated this problem substantially.

B. Other Filtering Methods

Before turning to the non-linear MP filter, we considered two methods that are commonly used to smooth out measurement error, thresholds and exponential averaging. We also

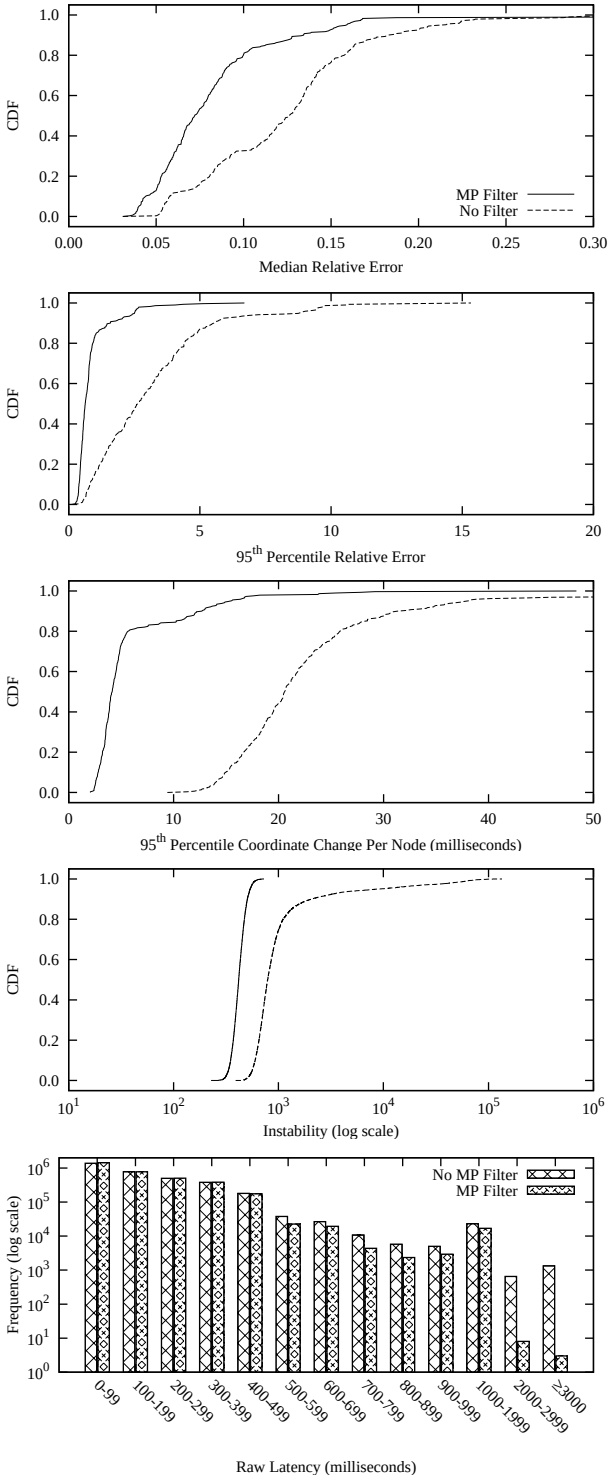

Fig. 5. Cumulative distributions of relative error (accuracy) and coordinate change (stability). The top two graphs show the median and 95^{th} percentile relative error for each node, respectively; thus, some nodes commonly experience several times more error than others. The third graph portrays that using the MP filter cuts instability per node in half for most nodes. The fourth graph shows a CDF of aggregate coordinate change per second (stability). With the MP filter, each node moves by a little more than one millisecond per second. Without the filter, spurious observations throw off all nodes' coordinates, resulting in a long tail. The filter improves global stability in the worst case by three orders of magnitude. The bottom graph shows how the MP filter only trims the problematic observations off of the end of the latency measurements, leaving the remainder of the distribution intact. The histogram is of the four hour subsection of the trace.

TABLE I
EXPONENTIALLY-WEIGHTED HISTORIES

Filter	Median Relative Error	Instability
MP Filter	0.07 (−42%)	415 (−47%)
No Filter	0.12 (0%)	783 (0%)
$\alpha = 0.02$	0.27 (+125%)	490 (−37%)
$\alpha = 0.10$	2.48 (+1960%)	1907 (+143%)
$\alpha = 0.20$	5.70 (+4650%)	3783 (+383%)

examined a confidence building method specific to Vivaldi, which would appear to increase coordinate convergence rates. Contrary to our initial expectations, these methods had negligible impact on accuracy or stability, and made conditions worse in some circumstances.

Thresholds. Prior to examining the latency distribution, we first considered using fixed threshold to discard extreme values. Dropping all values above a threshold is a simple method, with the added benefit that it requires no state. Given the distribution of the entire trace (shown in Figure 2), this method also removes the most extreme outliers, smoothing the process slightly. However, each link tended to show its own set of outliers: most links exhibited heavy tails, but the centering and length of the tail was different. For example, a cut-off that might work for the general distribution would do nothing for outliers in the link shown in Figure 3, where the common case is less than $100ms$. Early in our exploration, we tried several thresholds before moving to more complex techniques; we found only minimal stability and accuracy improvement when used in isolation.

EWMA. A commonly used filter to smooth jittery data is the exponentially-weighted moving average. It captures a distribution's general trend by including all previous observations and giving them an exponentially-declining weight: $v_{t+1} = \alpha \times s + (1-\alpha) \times v_t$, where v_t is the current value of the filter and v_{t+1} is the value after including observation s. The filter's behavior is controlled by one parameter, $0 < \alpha \leq 1$, which determines how much weight is given to the current observation.

We added a per-link EWMA to our simulator with the goal that it would capture changes in network conditions and dampen the outliers we had seen. We used conventional values for α of 0.10 and 0.20 and measured the same four hour section of the trace as in previous experiments. Table I shows the median value of the distribution of median relative error and stability when nodes use an EWMA filter with differing values of α, as compared to using no filter and using the MP filter. The data show that even when an unconventionally low value for α is used, 0.02, smoothing with an EWMA still results in lower accuracy than using no filter at all. The outliers are not signifying a trend an EWMA should capture, but instead should simply be discarded.

Confidence Building. The third potential improvement to Vivaldi is particular to the algorithm itself. Links with very low latency can prevent nodes from becoming confident in their coordinates. This occurs when the true latency between two nodes is beneath the precision of our latency measuring tools. When we first ran Vivaldi on our local cluster without the MP filter, we saw a fairly Normal spectrum of latency observations between 0.4 and $1.2ms$, and then a tail of 5% of the observations above $1.2ms$. Because the measurements used UDP and because the machines had no other load, we attributed the spread to context switches and background processes running on the machines. In essence, these observations were below our software's ability to detect them accurately.

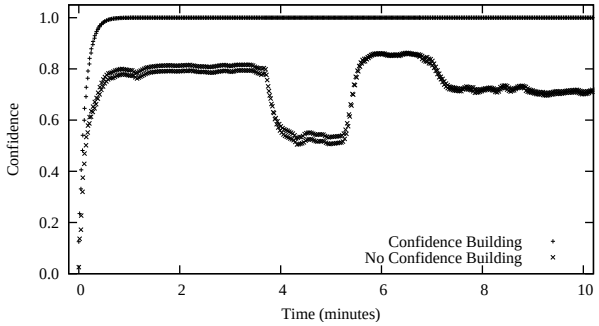

Fig. 6. By allowing for measurement error on low-latency links, nodes in the same cluster can gain confidence in their coordinates. However, in a wide-area network, suppressing spurious, high latency observations has much greater impact than precise measurement of low latency ones.

When run on a cluster with low latency, this jitter has an adverse effect on the Vivaldi algorithm. It results in high relative error (Figure 1, line 2) which in turn adversely affects the update in confidence (line 4). For example, if two nodes currently have confidence 0.5, and the sampler believes its neighbor is $1ms$ away in the coordinate space, a single sample of $3ms$ will reduce confidence by almost 5%.

To solve this problem, we introduced a margin of error that was allowed for each sample, a method we call *confidence building*. If the expected and actual measurement were within this margin of error, we considered them equal. Because we found our measurement error rarely exceeded three milliseconds (0.05%), we set the threshold to this value. This simple mechanism dramatically increased confidence in a low-latency environment.

To examine the effects of *confidence building*, we ran an experiment with three nodes on our local cluster. They computed their coordinates by choosing one node to sample every second, and we examined how Vivaldi performed with and without allowing for measurement error. Figure 6 shows how *confidence building* affects one node's confidence over a ten minute interval. Using *confidence building*, the node maintains 100% confidence after start-up. Without it, confidence wavers around 75%. There appear to be two lines with *No Confidence Building* because the the node's confidence changes slightly with each measurement to its two neighbors. Confidence followed the same pattern whether or not the MP filter was used in this environment; thus, the filter does not alleviate the confidence problem on a low latency network.

We conjectured that when several nodes participating in a large-scale network were co-located in the same subnet, they would reinforce each other's coordinates, essentially creating a confident reference point for other nodes. Surprisingly,

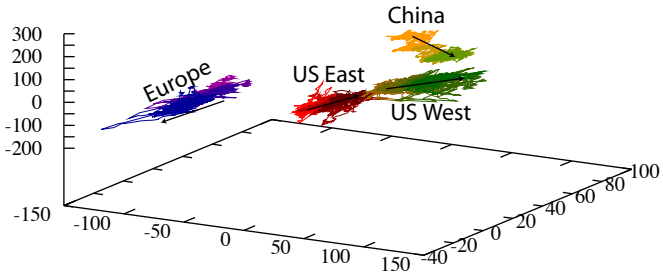

Fig. 7. Coordinates do not necessarily shift in the same direction over time, nor do they rotate about the origin. We show how four node's coordinates change over a three hour period. US West, US East, and China have shifted nearer to one another and the node in Europe has a higher latency to all three.

confidence building only had a small impact on the externally-visible metrics of accuracy and stability when run on a higher latency network. When using an MP filter, it further improved median relative error by 8.8% and stability by only 2.3%.

These results suggest that *confidence building* might be a useful technique if Vivaldi were run on a small cluster. However, network coordinates are primarily useful for large-scale networks, rendering *confidence building* and other efforts to improve the precision of small measurements, including kernel timestamping, less important than eliminating large spurious observations.

V. Updating Application-level Coordinates

Our use of the MP filter greatly improved the stability and accuracy of a set of network coordinates. As Figure 5 showed, use of the filter clipped the heavy tail of instability. However, the system's coordinates are still changing at about $500ms/sec$. For an application using network coordinates, is all this movement necessary? Instead of being notified about slight changes in coordinates with every observation, most applications would prefer to be notified only when a *significant* change occurs. By designing the coordinate subsystem as a black box that only signals when there is significant change, we can limit application updates that, in turn, limit unnecessary application-level work. For the application we developed, a coordinate change could initiate a cascade of events, culminating in one or more heavyweight process migrations. If the systems' coordinates have not changed significantly, there is no reason to begin this process. Of course, some applications would prefer a constant update: the subsystem should output both a system-level coordinate, $\overrightarrow{c_s}$, and an application-level one, $\overrightarrow{c_a}$. Those in the former category would use $\overrightarrow{c_a}$ and the latter $\overrightarrow{c_s}$.

Before considering how and when to update $\overrightarrow{c_a}$, we must ask: is it necessary to update $\overrightarrow{c_a}$ at all? That is, after some time, do coordinates cease to change relative to one another, merely rotating about an axis, oscillating, or otherwise remaining stationary? The answer is no: coordinates do change, reflecting changes in the underlying network even over relatively short time-scales. We illustrate this change in Figure 7 by showing how four nodes' coordinates vary over time. The nodes are from four distinct regions. Their coordinates move in a consistent direction over a three hour period, neither rotating nor remaining within one area. Instead, this example portrays that $\overrightarrow{c_a}$ should be updated over time to sustain accuracy.

The fact that $\overrightarrow{c_a}$ must be updated suggests a trade-off between the drawback of changing $\overrightarrow{c_a}$, which induces (perhaps unnecessary) application-level work and $\overrightarrow{c_a}$'s accuracy. Our goal is to shift the line in Figure 5 (bottom) to the left, increasing stability, without moving the line in Figure 5 (top) to the right, increasing error.

We examined four heuristics that each attempt to update $\overrightarrow{c_a}$ at appropriate times: dampening application updates while retaining the MP filter's low relative error. Two are based on simple thresholds and two on sliding windows of previous c_s coordinates. Before explaining the heuristics, we explain how we transform streams of system-level coordinate updates into two sets that can be tested for significant coordinate change.

A. Detecting Change with Windows

In the context of streams of samples entering a database, Ben-David, Gehrke, and Kifer propose an algorithm to detect when the stream has undergone a significant change [11]; their algorithm is similar to one proposed by Kleinberg for detecting word bursts in text streams [12]. The kernel of their idea is to divide a single data stream $S = \{s_0, s_1, \ldots, s_n\}$ into two sets, $W_s = \{s_0, \ldots, s_k\}$ and $W_c = \{s_{n-k}, \ldots, s_n\}$, that can be compared for statistically significant change using one of a handful of standard techniques (*e.g.*, rank-sum).

Initially, both W_s and W_c (*start* and *current*, respectively) are empty. As each element s_i arrives, it is added to W_s and W_c until they are both of size k. When this size is reached, no more elements are added to W_s, and W_c slides to add s_i and drop s_{i-k-1}. With each new element, the sets are tested for difference. When the two sets are declared to be different, a *change point* is said to have occurred. At this point, both windows W_s and W_c are cleared and the process begins again. By creating two distributions out of the single stream, they produce sets that can be compared for difference using well-known statistical tests. The well-known tests Ben-David *et al.* examine in their work, however, are all for one-dimensional data. The two tests we employed for multi-dimensional data are heuristics Energy and Relative below.

B. Application Update Heuristics

Now that we have explained how the two sliding window algorithms turn the streams of coordinates into two sets, we present four heuristics that each attempt to increase stability in application-level coordinates without decreasing their accuracy:

System. If the change in $\overrightarrow{c_s}$ from one observation to the next is greater than a threshold τ, update $\overrightarrow{c_a}$. Thus, if

$$\|\overrightarrow{c_{s(t)}} - \overrightarrow{c_{s(t-1)}}\| > \tau,$$

let $\overrightarrow{c_a} = \overrightarrow{c_s}$. This heuristic is simple but suffers from a pathological case: many changes just under the threshold

might occur, which would lead to high error. Note that relative error in this context is

$$\epsilon_a = \frac{|\ \|\overrightarrow{c_{a_i}} - \overrightarrow{c_{a_j}}\| - l_{ij}\ |}{l_{ij}}.$$

APPLICATION. If the application's idea of the coordinate has strayed too far from the system's, notify the application. More precisely, if

$$\|\overrightarrow{c_a} - \overrightarrow{c_s}\| > \tau,$$

let $\overrightarrow{c_a} = \overrightarrow{c_s}$. This heuristic is a simple way of expressing that an update should occur if a drift in one direction occurs; it permits oscillations beneath τ.

RELATIVE. This is the first of our two window-based heuristics. Here we measure the local relative distance as compared with our nearest known neighbor r and update the application if the change is larger than an error ϵ_r. RELATIVE averages each of its sets of coordinates by taking their centroid $\mathcal{C}(W)$. It computes, if

$$\frac{\|\mathcal{C}(W_s) - \mathcal{C}(W_c)\|}{\|\mathcal{C}(W_s) - \overrightarrow{r}\|} > \epsilon_r,$$

let $\overrightarrow{c_s} = \mathcal{C}(W_c)$. This heuristic exhibits three good properties: updates are relative to the node's locale, computing the centroid is inexpensive, and $\mathcal{C}(W_s)$ can be cached. The approximate nearest neighbor is learned through a comparison with each latency sample, where the node learns $\overrightarrow{x_j}$.

ENERGY. The last heuristic uses a statistical test that specifically measures the Euclidean distance between two multi-dimensional distributions [26]. It is based on the *energy* distance $e(A, B)$ between two finite sets $A = \{\overrightarrow{a_1}, \ldots, \overrightarrow{a_{n_1}}\}, B = \{\overrightarrow{b_1}, \ldots, \overrightarrow{b_{n_2}}\}$:

$$e(A,B) = \frac{n_1 n_2}{n_1 + n_2}\left(\frac{2}{n_1 n_2}\sum_{i=1}^{n_1}\sum_{j=1}^{n_2}\|\overrightarrow{a_i} - \overrightarrow{b_j}\| - \frac{1}{n_1^2}\sum_{i=1}^{n_1}\sum_{j=1}^{n_1}\|\overrightarrow{a_i} - \overrightarrow{a_j}\| - \frac{1}{n_2^2}\sum_{i=1}^{n_2}\sum_{j=1}^{n_2}\|\overrightarrow{b_i} - \overrightarrow{b_j}\|\right)$$

Using this statistic, we can determine the divergence of the two windows. If

$$e(W_s, W_c) > \tau,$$

let $\overrightarrow{c_a} = \mathcal{C}(W_c)$. While computing this heuristic is more computationally intensive than RELATIVE, the difference is negligible for the small windows we used.

C. Summary of Application-Update Results

To examine how these four heuristics affected stability and accuracy from an application's viewpoint, we implemented them in our simulator and used the same trace of node latencies. In particular, we wanted to see how the window size and threshold parameters affected these metrics.

The following summarizes the results of our comparison of application-update heuristics:

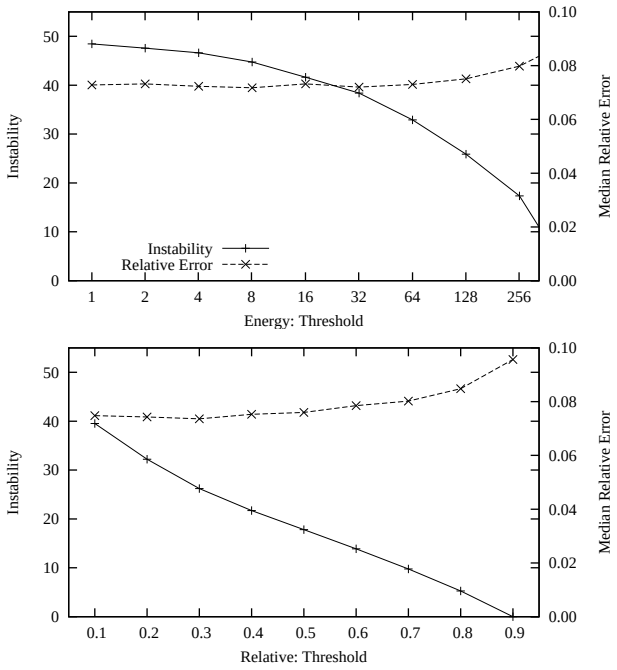

Fig. 8. Instability and Median Relative Error for varying threshold with RELATIVE and ENERGY.

- As expected, increasing the threshold required for application update increases stability but also decreases accuracy. The window-based heuristics succeed in substantially increasing stability before any significant decline in accuracy begins.
- Large windows, *e.g.,* between 32 and 512 samples, improve both stability and accuracy. Very large windows, however, cause too few updates to occur, decreasing accuracy.
- The heuristics that do not use windows can increase stability only at the immediate expense of accuracy and are not robust to minor parameter changes.

D. Window-based Heuristics

Because the window-based heuristics, RELATIVE and ENERGY, are more complex, with their two parameters of window size and threshold, we examined their behavior first. We conjectured that, as the threshold for update increased, fewer updates of $\overrightarrow{c_a}$ would occur, leading to greater stability and perhaps reduced accuracy.

To examine how the thresholds τ and ϵ_r affect ENERGY and RELATIVE, respectively, we ran an experiment where we varied the value of the threshold and kept window size constant. We recorded accuracy and stability and Figure 8 shows the median for both the distribution of median relative error per node and of instability. The results summarize the last two hours of the four hour trace, as in previous experiments.

The data establish that RELATIVE exhibits a near-linear increase in stability with increasing threshold. Thus, as RELATIVE requires more and more movement relative to the distance to the nearest neighbor, updates steadily decline. The increase in ENERGY's stability is curved but has no knee: it too exhibits

a measured decline in coordinate change as the threshold to update increases. Both heuristics fall in the same range of relative error, with ENERGY exhibiting a more gradual decline as thresholds increase. However, the decline in accuracy for both heuristics does not expend a substantial increase in stability, especially for RELATIVE, where instability is cut in half without any noticeable reduction in accuracy. Accuracy begins to decline for ENERGY after $\tau = 8$ and for RELATIVE after $\epsilon_r = 0.3$. These are the most conservative parameters that still grant an increase in stability, with 8% for RELATIVE and 34% for ENERGY. We kept window size at 32 for this experiment.

Our second experiment with the window-based heuristics was to establish reasonable boundaries for window size. Unlike the per-link MP filter, using a large window is acceptable because windows are appended to with every observation, regardless of the link. However, similar to using a large filter, a trade-off exists in which very large windows are slow to react to true changes in underlying network conditions.

We ran an experiment in which we kept the threshold for application-update constant while we varied window size exponentially. We monitored accuracy and stability as before, and also observed how frequently application updates occurred over time. This last number — that is, the number of times $\overrightarrow{c_a}$ is changed per unit time — is interesting because even though stability might be increased, it might not necessarily correlate with a decline in application notifications. Instead, stability could be increasing through smaller updates that occur at the same frequency. Because a cost exists in notifying an application with a coordinate change, we wanted to ensure that both instability and update frequency were decreasing. In Figure 9, we show the same metrics as the previous experiment together with the percent of the 269 nodes that changed their values for $\overrightarrow{c_a}$ each second. The data show that not only do large windows ($\approx 2^5 - 2^9$) modestly improve accuracy, but also they result in a steady increase in stability and decline in update frequency. Across a wide range of window sizes, updates are both less frequent and cause less movement in aggregate, achieving two of the goals of the application-update heuristics. At a window size of 128 for example, RELATIVE's median relative error is 7%, its instability $5ms/sec$, while causing only 1% of the nodes to be updated per second. This is a 42% increase in accuracy and a two orders-of-magnitude improvement in stability compared to the original algorithm. Because all large window sizes afforded a substantial improvement in the metrics, we chose the smallest of these, 32, to make a conservative comparison with the window-less heuristics and to use in our PlanetLab implementation. We used the threshold values gathered from the previous experiment.

E. Windowless Heuristics

The window-based heuristics have the disadvantage that they are slightly more complex than the windowless ones, SYSTEM and APPLICATION, and that they require more state. Using the parameters we established for window size from the previous experiment, we compared all four heuristics as we varied the update threshold. Unlike ENERGY and RELATIVE the windowless heuristics could only directly trade off accuracy for stability and had a limited "sweet spot," one which might change with a different trace.

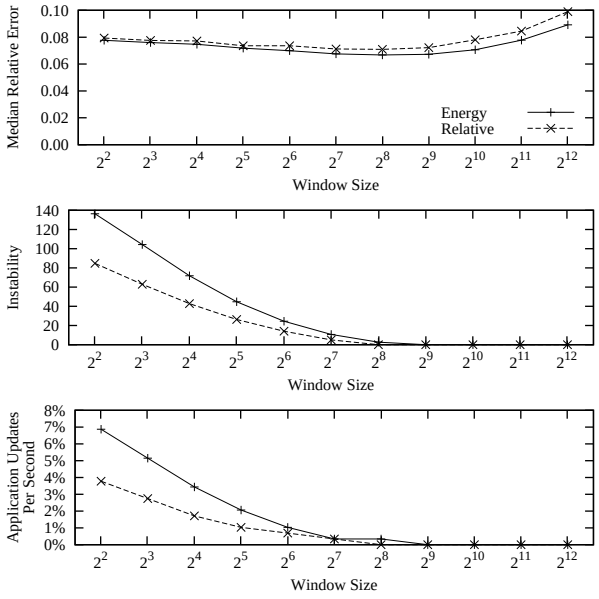

Fig. 9. Median Relative Error, Instability and Application Updates per Second with varying window size for RELATIVE and ENERGY.

We show the same metrics, median relative error, and instability, as we vary threshold in Figure 10. At low thresholds, when $\overrightarrow{c_a}$ is updated after only a small movement from its previous value, SYSTEM's and APPLICATION's performance remain similar to the raw MP filter. With a large threshold, $\overrightarrow{c_a}$ is rarely updated, leading to high error. Only at $\tau = 16$ do the two heuristics perform in the same range as the window-based ones. Because tipping in either direction results in poor performance on one of the metrics, we conclude the added complexity and state of using one of the window-based heuristics is worthwhile.

F. Comparison to the Raw MP Filter

Our primary goal in introducing the application-level heuristics was to further improve stability while maintaining accuracy. In Figure 11, we show how the two window-based heuristics achieve that goal. Using the parameters established above, accuracy remains unchanged while RELATIVE and ENERGY shift the entire distribution of coordinate updates into a more stable regime.

G. Discussion

Application-level accuracy and stability depend on both knowing when to update $\overrightarrow{c_a}$ and what to set it to. A substantial component of the success of the two window-based heuristics is their setting $\overrightarrow{c_a} = \mathcal{C}(W_c)$. One could argue that a simple threshold scheme might achieve similar performance if it too used the centroid of a collection of recent system-level coordinates. However, while it is true that all RELATIVE and ENERGY do is set $\overrightarrow{c_a}$ to the centroid of recent values for c_s,

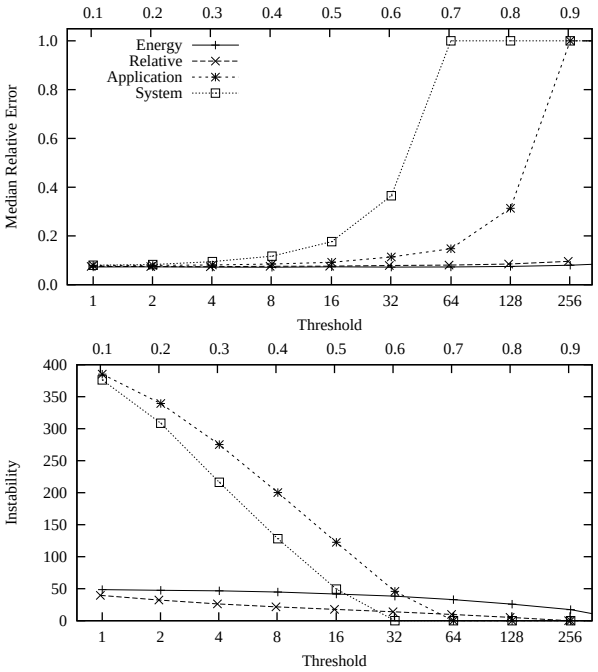

Fig. 10. Effect of varying threshold for all four heuristics. The window-based heuristics maintain high accuracy and stability. The simple threshold-based ones can only trade-off accuracy for stability and are much more sensitive to changes in the threshold parameter.

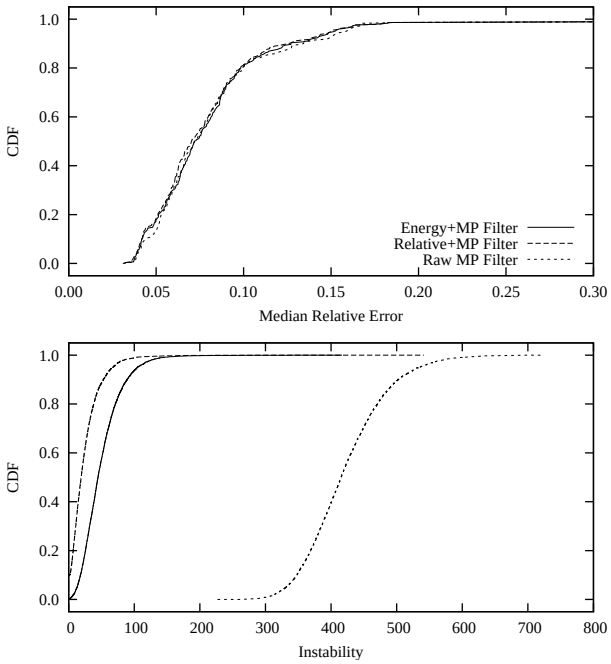

Fig. 11. Comparison of application-level suppression to Raw MP Filtering. Both window-based heuristics, RELATIVE and ENERGY succeed in keeping relative error low while greatly increasing coordinate stability.

achieving the proper *rate* for these updates — knowing when to change — is a property simple thresholds have difficultly performing.

To test this claim, we modified APPLICATION to set $\overrightarrow{c_a}$ to be the centroid of a window of the past 32 coordinates (the same size that ENERGY and RELATIVE use above). In our experiment, we varied the threshold at which updates were made and again monitored accuracy and stability. As the data in Figure 12 portray, this combined APPLICATION/CENTROID is more stable than APPLICATION and SYSTEM but, like the two window-less heuristics, it is not robust against slight changes in parameters and has high stability only at the expense of good accuracy.

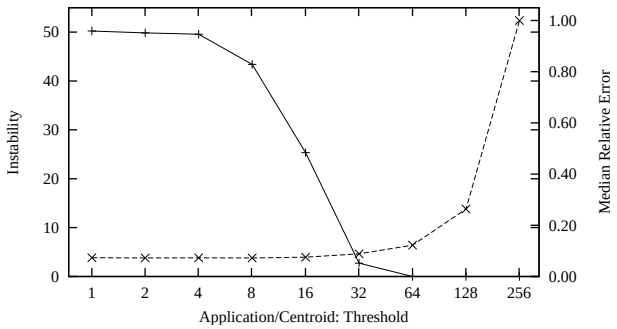

Fig. 12. Instability and Median Relative Error with varying threshold for APPLICATION/CENTROID.

VI. PLANETLAB EXPERIMENT

In order both to verify our simulator and to confirm that our findings were not limited to our latency trace, we implemented a version of Vivaldi that could be run on a real network. This version uses application-level UDP pings as input, the same as our trace. Each node started with a small neighbor set and gossiped one address with every sample. Nodes sampled from their neighbor set in round-robin order at five second intervals. We added the MP filter and the ENERGY application-level update heuristic to our implementation. We used a window of 32 and $\tau = 8$ as suggested by the parameter space exploration in simulation.

In order to ensure a valid comparison between running Vivaldi with our enhancements and without, we ran them on the same set of PlanetLab nodes at the same time, using different ports. One set of nodes used the MP filter and one did not; both used ENERGY. Because each node outputted $\overrightarrow{c_s}$ and $\overrightarrow{c_a}$ with each sample, we could monitor the effects of the filter and the update heuristic separately. We ran this pair of coordinate systems for four hours on 270 PlanetLab nodes on June 24, 2005.

The results of the real-world experiment confirm those of our simulations. We show the relative error and stability for the second half of the experiment in Figure 13. The data show that the MP filter reduces error and instability and the application-update heuristic further increases stability. We also examined how the MP filter and update heuristic affected these metrics over time, shown in Figure 14. The data show that after a half hour convergence period, using the MP filter and ENERGY result in a much smoother and more accurate metric space on a real wide-area network. The data confirm that both enhancements have distinct effects on the two metrics and that both are required for a stable and accurate space from an application perspective.

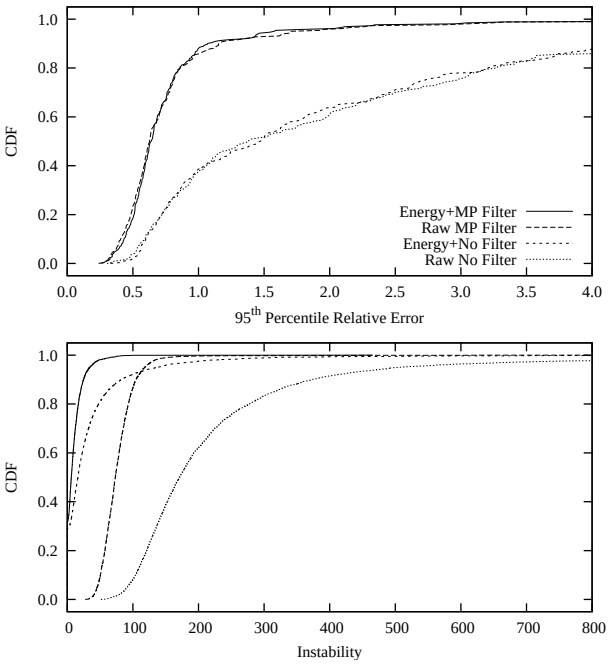

Fig. 13. Cumulative distribution of relative error and instability of Vivaldi running on PlanetLab. The data show that with the MP filter only 14% of the nodes experienced a 95^{th} percentile relative error greater than one, while 62% of those without the filter did. ENERGY dampened the filter's updates: 91% of the time it fell below even the minimum instability of the raw filter. The enhancements combine to reduce the median of the 95^{th} percentile relative error by 54% and of instability by 96%.

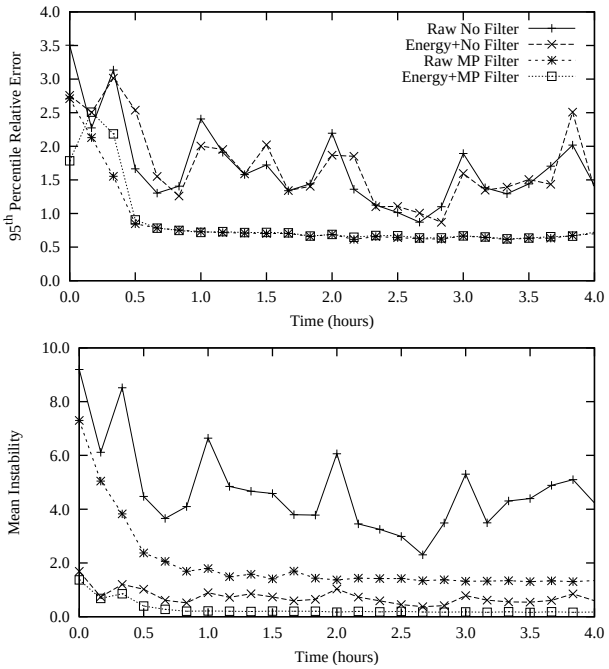

Fig. 14. Relative error and instability vary with time on PlanetLab. The data points are the median error and mean instability for ten minute intervals.

After a close examination of any coordinate disruptions during the PlanetLab experiment, we discovered a source of much of the worst error. Most real-time low pass filters add delay in order to incorporate future values. Our MP filter outputted a value for every input, regardless of the history length: it produced the p^{th} percentile of the current state it was storing. Thus a pathological case occurs when an extreme outlier is the first observation for a particular link: even with the filter, this observation is what is used. In fact, this was the case for the five largest node displacements in the PlanetLab experiment and the echoes of these disruptions often continued for minutes. To compensate for this, Vivaldi could wait until a sufficient number of samples are in the filter.

In simulation, we experimented with waiting until the second sample on a link to return an observation. This greatly reduced early instability, but, because our set of nodes was constant, had only limited impact after start-up. In a long-running system where nodes periodically enter and leave, adding a delay to the filter would increase its robustness against these pathological cases at only a small cost.

VII. Related Work

A. Synthetic Network Coordinates

Since Ng and Zhang provided the first in-depth examination of how to embed inter-node latencies in a metric space [15], a series of different approaches have emerged. In their initial work, called Global Network Positioning, a coordinate space was built in two stages: first, a collection of well-known *landmarks* placed themselves in a vector space through all-pairs ping measurements; second, each joining node measured its distance to all of the landmarks and picked a coordinate that minimized the error to all of them. This approach does not allow for a smooth evolution of the space over time, nor is it decentralized. However, it did establish that, even with the error induced by triangle inequality violations, a high-quality space was possible. Lighthouses [18] Mithos [27], and NPS [16] extended the landmark approach by using multiple local coordinate systems, by building the space through preferring to measure nearby neighbors, and through a hierarchical architecture, respectively. More recently, Costa *et al.* developed PIC, another landmark scheme, which runs a Simplex solver on each node to minimize error [3]. PIC readjusts coordinates through periodically re-running this solver process and includes a test to defend its coordinate system against malicious participants. Cox *et al.* initially proposed the decentralized Vivaldi algorithm we discuss here [4] and Dabek *et al.* later improved its accuracy in two-dimensions with *height*, which was intended to explicitly capture the latency to a high speed link [5]. Shavitt and Tankel's Big-Bang Simulations is an embedding technique similar to Vivaldi, although it models a potential force field instead of a mass-spring system [25]. Kleinberg has developed a theoretical grounding for network embeddings, analyzing how to embed coordinates with arbitrarily low errors [13].

Network embeddings were developed partially in response to the growing interest in topologically-efficient overlay routing. CAN's multi-dimensional space [21], in particular, has motivated work on network-aware overlays and on using a node's network coordinates as its logical CAN coordinate [22], [31], [29]. In recent theoretical work, Abraham and Malkhi

have examined routing strategies made possible through the existence of network embeddings [1]. As network embeddings have become better understood, work has surpassed using them merely to route; current work explores how they can be used for operator placement in distributed streaming databases and for solving the distributed approximate k-nearest neighbors problem [19].

In contrast, other work has tried to solve the same set of problems, including the k-nearest neighbor problem, without establishing a coordinate space, arguing that their maintenance is a burden and that these coordinate spaces exhibit higher error than a customized mechanism. In essence, this class of work solves the neighbor and routing problems *reactively*, through a spike in activity in response to an application-driven demand, while a long-running coordinate space solves them pro-actively. Meridian, for example, finds the nearest overlay node (*i.e.*, one running Meridian) to an arbitrary point in the Internet through a large set of pings in direct response to an application-level request [28]. In the same vein, Shanahan and Freedman examine the efficacy of network embeddings for finding nearby servers for unmodified clients [24]. The choice between solving these problems reactively or pro-actively appears to be an application-specific decision.

B. Stabilizing Vivaldi

We used Szekely and Rizzo's *energy* statistics as one heuristic to find the distance between the start and current coordinate windows [26]. Rubinfeld and Servedio provide an alternate algorithm for determining the ϵ-distance in Euclidean space for two distributions [23]. However, their tests are more focused on high dimensions and reducing the number of samples required for comparison. In recent work, Zech and Aslan independently proposed a test statistic, also called *energy*, which differs from the statistic we used in its inclusion of a problem-dependent scaling function embedded within the statistic [30].

In another effort to stabilize Vivaldi, de Launois *et al.* modify the algorithm to prevent oscillations in the presence of triangle inequalities [6]. They introduce a factor that asymptotically dampens the weight given to each new measurement, regardless of its source. While this factor does mitigate oscillations, it prevents the algorithm from adapting to changing network conditions as the pull of new measurements approaches zero.

VIII. Conclusion

In a real-world deployment, no fixed, single-valued latency matrix exists. Instead, nodes see a stream of latency values along each link. When these raw values are used to embed hosts into a metric space, the coordinate system they create is fragile.

Common techniques, *e.g.*, excluding "large" values and using exponentially-weighted filters do not create a useful set of latencies. Instead, a short non-linear low pass filter, the moving percentile filter, both removes extreme values and is agile enough to allow the output signal to accurately reflect changes in the underlying network. Additionally, the benefit of using more precise measurement tools is small relative to eliminating signal extrema with a low pass filter.

We introduced the distinction between system- and application-level coordinates and examined the effect of four heuristics that determine how and when to update the application-level coordinate. The two heuristics, ENERGY and RELATIVE, that used a change-detection algorithm based on sliding windows best determined when to make the update. Additionally, using the centroid of a collection of recent coordinates set the application-level coordinate to a highly accurate value. We confirmed the results from our simulations with an implementation on PlanetLab.

Recalling Vivaldi's two useful properties, scalability and continuous refinement, our set of techniques do not fundamentally alter them in practice. The MP filter is short enough to permit on-going adjustments and does not affect scalability. The application-level update heuristics, too, allow the algorithm to function as before, only with increased stability for applications.

Acknowledgments

The authors would like to thank Michael Mitzenmacher for helpful discussions.

References

[1] I. Abraham and D. Malkhi. Compact routing on euclidian metrics. In *23rd Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, July 2004.

[2] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[3] M. Costa, M. Castro, A. Rowstron, , and P. Key. PIC: Practical Internet Coordinates for Distance Estimation. In *International Conference on Distributed Computing Systems*, Tokyo, Japan, March 2004.

[4] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical distributed network coordinates. In *Second Workshop on Hot Topics in Networks*, November 2003.

[5] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM*, Portland, OR, Aug. 2004.

[6] C. de Launois, S. Uhlig, and O. Bonaventure. A Stable and Distributed Network Coordinate System. Technical report, Universite Catholique de Louvain, December 2004.

[7] T. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling. p2psim. `http://www.pdos.lcs.mit.edu/p2psim/`.

[8] S. Husen, R. Taylor, R. Smith, and H. Healser. Changes in geyser behavior and remotely triggered seismicity in Yellowstone National Park produced by the 2002 M7.9 Denali fault earthquake. *Geology*, 32:537–540, 2004.

[9] P. Indyk. Algorithmic applications of low-distortion geometric embeddings. In *42nd Annual Symposium on Foundations of Computer Science*, Las Vegas, Nevada, 2001.

[10] KaZaA. `http://www.kazaa.com`.

[11] D. Kifer, S. Ben-David, and J. Gehrke. Detecting Change in Data Streams. In *Thirtieth International Conference on Very Large Data Bases*, Toronto, Canada, August 2004.

[12] J. M. Kleinberg. Bursty and hierarchical structure in streams. In *Eighth International Conference on Knowledge Discovery and Data Mining*, Edmonton, Alberta, Canada, July 2002.

[13] J. M. Kleinberg, A. Slivkins, and T. Wexler. Triangulation and embedding using small sets of beacons. In *45th Symposium on Foundations of Computer Science*, Rome, Italy, October 2004.

[14] A. W. Moore and J. W. Jorgenson. Median Filtering for Removal of Low-Frequency Background Drift. *Analytic Chemistry*, 65:188, 1993.

[15] E. Ng and H. Zhang. Predicting Internet Network Distances with Coordinate-based Approaches. In *INFOCOM*, New York, NY, June 2002.

[16] E. Ng and H. Zhang. A Network Positioning System for the Internet. In *USENIX*, Boston, MA, June 2004.

[17] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *First Workshop on Hot Topics in Networks*, October 2002.

[18] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. Lighthouses for Scalable Distributed Location. In *Second International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[19] P. Pietzuch, J. Ledlie, J. Shneidman, M. Welsh, M. Seltzer, and M. Roussopoulos. Network-Aware Operator Placement for Stream-Processing Systems. Technical Report TR-04-05, Harvard University, June 2005.

[20] P. Pietzuch, J. Shneidman, M. Roussopoulos, M. Seltzer, and M. Welsh. Path Optimization in Stream-Based Overlay Networks. In *First International Workshop on Networking Meets Databases*, Tokyo, Japan, April 2005.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *SIGCOMM*, August 2001.

[22] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *INFOCOM*, New York, NY, June 2002.

[23] R. Rubinfeld and R. Servedio. Testing monotone high-dimensional distributions. In *Symposium on the Theory of Computing*, Baltimore, MD, May 2005.

[24] K. Shanahan and M. Freedman. Locality Prediction for Oblivious Clients. In *Fourth International Workshop on Peer-to-Peer Systems*, Ithaca, NY, February 2005.

[25] Y. Shavitt and T. Tankel. Big-Bang Simulation for embedding network distances in Euclidean space. In *INFOCOM*, San Francisco, CA, June 2003.

[26] G. Szekely and M. Rizzo. Testing for Equal Distributions in High Dimension. *InterStat*, 5, November 2004.

[27] M. Waldvogel and R. Rinaldi. Efficient topology-aware overlay network. In *HotNets-I*, Princeton, NJ, Oct. 2002.

[28] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *SIGCOMM*, Philadelphia, PA, August 2005.

[29] Z. Xu, C. Tang, and Z. Zhang. Building topology-aware overlays using global soft-state. In *22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[30] G. Zech and B. Aslan. A multivariate two-sample test based on the concept of minimum energy. In *PHYSTAT*, Stanford, CA, September 2003.

[31] B. Zhao, Y. Duan, L. Huang, A. Joseph, and J. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *First International Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002.